DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 0
Iniciación a Java

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (09/2019)

UD0 – Iniciación a Java

ÍNDICE

1. Motivación

Si estás estudiando Acceso a Datos, como parte del segundo curso del Ciclo Superior de Desarrollo de Aplicaciones Multiplataforma, seguro que ya tienes soltura programando, pero quizá no sea con el lenguaje Java, sino con C#, C++ o algún otro.

Por eso, aquí tienes un repaso básico de Java, que te ayude a poder conocer las principales estructuras del lenguaje y compararlas con el lenguaje o lenguajes que ya conoces. Si ya has programado en Java, pero no recientemente, te interesará al menos hacer los ejercicios que encontrarás al final de este tema.

2. Esqueleto básico

El primer programa va a ser el clásico “Hola Mundo”, que escribe ese texto en pantalla. La precaución que no debes olvidar es que el nombre del fichero debe coincidir con el nombre de la “clase” que representa el programa (por ejemplo, este programa deberá estar en un fichero llamado “HolaMundo.java”):

```
// Fichero HolaMundo.java
// Compilar con: javac HolaMundo.java
// Ejecutar con: java HolaMundo

public class HolaMundo {
  public static void main(String [] args) {
    System.out.println("Hola mundo.");
  }
}
```

Algunas recomendaciones de **buenas prácticas** para facilitar la lectura del código y que así sea más fácilmente entendible y mantenible son:

- La documentación y/o comentarios
- La tabulación
- La nomenclatura

Es importante elegir un estándar de nomenclatura a la hora de programar. Estas convenciones o estándares de nomenclatura son un conjunto de normas para un lenguaje de programación específico y se recomiendan como buenas prácticas para facilitar la lectura del código y sea más fácilmente entendible y mantenible.

Veamos las más usadas:

- **Upper Case**: Todas las letras del identificador se encuentran en mayúsculas. En Java es la más usada para las constantes.
  Ejemplo: EJEMPLODENOMENCLATURA

- **Camel Case**: El nombre viene porque se asemeja a las dos jorobas de un camello, y se puede dividir en dos tipos:
  - **Upper Camel Case**, cuando la primera letra de cada una de las palabras es mayúscula. También denominado Pascal Case.
    En Java es el más usado para la nomenclatura de clases y tipos.
    Ejemplo: EjemploDeNomenclatura.
  - **Lower Camel Case**, igual que la anterior con la excepción de que la primera letra es minúscula.
    En Java es la más usada para las variables, objetos, métodos y funciones.
    Ejemplo: ejemploDeNomenclatura.
- **Snake Case**: Cuando cada una de las palabras, se separa por un guión bajo (_). Es común en los nombres de variables y funciones de lenguajes como C, aunque también Ruby y Python lo han adoptado. Como el Camel Case existen variedades, como todas las letras en mayúsuculas, denominado SCREAMING_SNAKE_CASE, que se utiliza para definir constantes.
  Ejemplo: ejemplo_de_nomenclatura.
- **Kebab Case**: Es igual que el Snake Case, esta vez, son guiones medios (-) los que separan las palabras. Su uso más común es de las urls.
  Ejemplo: ejemplo-de-nomenclatura

Por supuesto, hay más y sus usos son variados, como la notación húngara, l33t o leet, …

3. Compilar

Editor de Textos

Para escribir el código necesitaremos un editor de textos. **Notepad++** reconoce el lenguaje Java, así como también otros como HTML, CSS, SQL, …

Otros editores son **Sublime Text o Atom** que muestran además un árbol de las carpetas y archivos del proyecto.

En nuestro ejemplo usaremos **Notepad++**.

Compilador

En teoría, los pasos son sencillos, aunque en la práctica se pueden complicar un poco:

1. Teclear el fuente usando cualquier editor de texto.
2. Compilar con “javac”, por ejemplo:
   javac HolaMundo.java
3. Lanzar con “java”, por ejemplo:
   java HolaMundo

En la práctica pueden aparecer distintos problemas, como no tener Java instalado, que el nombre del fichero sea incorrecto o (en programas más complejos) no encontrar alguna de las bibliotecas que el programa necesite.

Para descargar e instalar JDK de Java Java accederemos a la web oficial:

https://www.oracle.com/technetwork/es/java/javase/downloads/index.html

Por defecto se instalará en "**C:\Program Files\Java**".

Para poderlo utilizar, tenemos dos opciones:
1. Indicar el path completo de java.exe y javac.exe
2. Añadir el path al path por defecto

En nuestro caso crearemos la carpeta en "**C:\JAVAPROG**" donde guardaremos todos los fuentes.

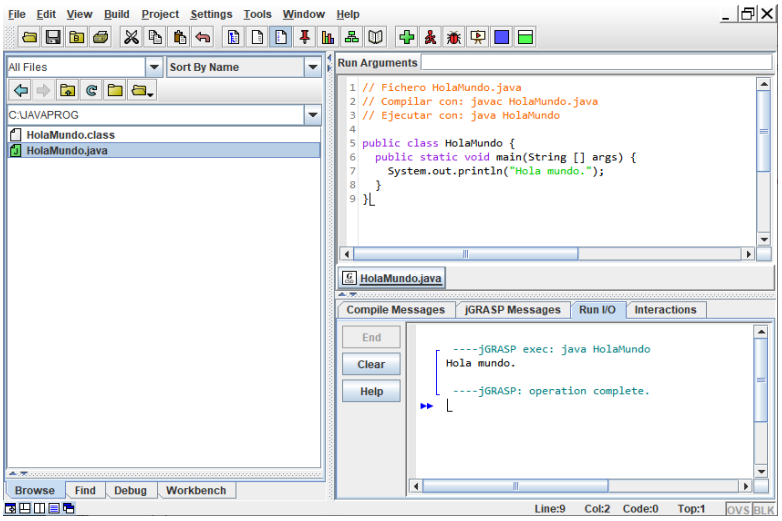
Con el editor crearemos el archivo **HolaMundo.java** indicado anteriormente.

Luego abriremos una venta de comandos:

```
// Para compilar
C:\JAVAPROG> "C:\Program Files\Java\jdk1.8.0_221\bin\javac" HolaMundo.java

// Comprobar que tenemos el fuente y el compilado
C:\JAVAPROG> dir /w
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: E264-EBA5

 Directorio de C:\JAVAPROG

[.]              [..]             HolaMundo.class   HolaMundo.java
               2 archivos            548 bytes

// Para ejecutar
C:\JAVAPROG> "C:\Program Files\Java\jdk1.8.0_221\bin\java" HolaMundo
Hola mundo.
```

IDE – Entornos de Desarrollo Integrado

Hay alternativas más avanzadas, que permiten ahorrar algo de trabajo al no tener que teclear siempre "**javac**" y "**java**", pero que no suelen venir preinstaladas en el sistema.



Nosotros usaremos para empezar **JGRASP**, un IDE gratuito que contempla lo imprescindible para poder editar, compilar y ejecutar programas Java.

Se puede descargar de su web oficial:

https://www.jgrasp.org/

Las características más importantes son:

- Simplicidad
- Dispone de Debugger
- Puede abrir los archivos java (con doble click) sin necesidad de crear proyectos

Como desventaja principal respecto a Netbeans o Eclipse indicar que:

- no está en español
- no es sensible a solucionar errores, ni dispone de ayuda sintáctica
- no incorpora automáticamente las librerías
- no dispone de herramientas de automatización de código como Maven o Graddle

4. Leer desde la consola

Si el usuario debe introducir datos, la forma más sencilla de conseguirlo es usando la clase **Scanner**, que tiene métodos como **nextInt**(), **nextFloat**(), **nextLine**() y **next**() (este último lee la siguiente cadena de texto y se detiene cuando encuentre un espacio o un avance de línea, mientras que "nextLine" es capaz de leer una cadena que contenga espacios intermedios y se detiene cuando llega al final de línea).

```
import java.util.Scanner;

public class Saludo1 {
  public static void main(String [] args) {

    Scanner entrada = new Scanner(System.in);
    System.out.print("Cuántas veces te saludo? ");

    int n = entrada.nextInt();
    for (int i=0; i<n ; i++)
      System.out.println("Hola mundo.");

  }
}
```

Al igual que ocurre en ocasiones en C y C++ (pero no en C#), puede haber problemas cuando se leen a la vez números y texto, porque "nextInt" deja el "avance de línea" (la pulsación de la tecla Intro) en el buffer del teclado, de modo que un "nextLine" posterior podría leer una cadena vacía, como en este caso:

```
import java.util.Scanner;

public class Saludo2 {
  public static void main(String [] args) {

    Scanner entrada = new Scanner(System.in);
    System.out.print("Cuántas veces te saludo? ");
    int n = entrada.nextInt();

    System.out.print("Y cual es tu nombre? ");
    String nombre = entrada.nextLine();

    for (int i=0; i<n ; i++)
       System.out.println("Hola " + nombre);

  }
}
```

Una alternativa simple es saltar ese avance de línea con un “nextLine” adicional, como en este ejemplo:

```
import java.util.Scanner;

public class Saludo2 {
  public static void main(String [] args) {

    Scanner entrada = new Scanner(System.in);
    System.out.print("Cuántas veces te saludo? ");
    int n = entrada.nextInt();
    entrada.nextLine();

    System.out.print("Y cual es tu nombre? ");
    String nombre = entrada.nextLine();

    for (int i=0; i<n ; i++)
       System.out.println("Hola " + nombre);

  }
}
```

5. Condiciones

La forma más sencilla de comprobar condiciones es con la orden “if”, acompañada quizás por la cláusula opcional “else”:

```
public class EjemploDeIf {
  public static void main(String [] args) {

    int x = 10;

    if (x == 70) {
      System.out.println("x vale 70");
    } else {
      System.out.println("x no vale 70");
    }

  }
}
```

Los símbolos con los que se indican las posibles condiciones son:

Operación	Símbolo
Mayor que	>
Mayor o igual que	>=
Menor que	<
Menor o igual que	<=
Igual que	==
Distinto de	!=

Y se puede enlazar varias condiciones con:

Operación	Símbolo
Y	&&
O	||
No	!

Cuando se trata de analizar varios posibles valores de una misma variable, puede resultar más cómodo usar la orden “switch”:

```
public class Switch1 {
  public static void main(String [] args) {

    int mes = 2;
    switch(mes) {
      case 1:
        System.out.println("El mes es Enero");
        break;
      case 2:
        System.out.println("El mes es Febrero");
        break;
      case 3:
        System.out.println("El mes es Marzo");
        break;
    }
  }
}
```

En su formato más completo, la orden “switch” permite que varios casos compartan código, así como indicar un “caso por defecto” que absorba todos los valores no indicados:

```
public class SwitchDefault {
  public static void main(String [] args) {

    int x = 5;
    switch(x) {
      case 1:
      case 2:
      case 3:
        System.out.println("El valor de x estaba entre 1 y 3");
        break;
      case 4:
      case 5:
        System.out.println("El valor de x era entre 4 o 5");
        break;
      case 6:
        System.out.println("El valor de x estaba era 6");
        break;
      default:
        System.out.println("El valor de x no estaba entre 1 y 6");
        break;
    }
  }
}
```

A partir de Java 7, también se puede usar “switch” para comparar cadenas de texto:

```
public class SwitchCadenas {
  public static void main(String [] args) {

    String nombre = "yo";
    switch(nombre) {
      case "uno":
        System.out.println("Hola, uno");
        break;
      case "yo":
        System.out.println("Hola, tú");
        break;
      case "otro":
        System.out.println("Bienvenido, otro");
        break;
      default:
        System.out.println("Nombre desconocido");
        break;
    }
  }
}
```

6. Bucles

Una condición se puede comprobar de forma repetitiva con “while”:

```
import java.util.Scanner;

public class While1 {
  public static void main(String [] args) {

    Scanner teclado = new Scanner(System.in);

    System.out.print("Introduce un cero: ");
    int dato = teclado.nextInt();

    while (dato != 0) {
      System.out.print("No era cero. Introduce cero: ");
      dato = teclado.nextInt();
    }
    System.out.println("Terminado!");
  }
}
```

Si la condición se debe comprobar al final (porque se desee dar al menos “una pasada”), se deberá usar “do-while”:

```
import java.util.Scanner;

public class DoWhile1 {
  public static void main(String [] args) {

    int password;
    Scanner teclado = new Scanner(System.in);

    do {
      System.out.print("Introduzca su password numérica: ");
      password = teclado.nextInt();
      if (password != 1234)
        System.out.println("No es válida.");
    } while (password != 1234);
  }
}
```

Y para contar, aunque se podría usar un “while”, es habitual emplear la orden “for”, que tiene un formato más compacto:

```
public class For1 {
  public static void main(String [] args) {

    int i;

    for ( i=1 ; i<=10 ; i++ ) {
      System.out.print("Hola ");
    }
  }
}
```

También se puede declarar la variable que actúa de contador dentro del propio “for”, para evitar reutilizar variables de forma no intencionada:

```
public class For2 {
  public static void main(String [] args) {

    for ( int i=1 ; i<=10 ; i++ ) {
      System.out.print("Hola ");
    }
  }
}
```

La orden “for” permite otro formato adicional en versiones recientes de Java, que veremos más adelante.

7. Tipos de datos básicos

Tenemos disponibles diversos tipos de datos numéricos, además de los “enteros normales” (int). La siguiente tabla resume su nombre, capacidad y (cuando se trata de números reales) precisión:

<table>
<tr><td rowspan="19">TIPOS DE DATOS EN JAVA</td><td></td><td>NOMBRE</td><td>TIPO</td><td>OCUPA</td><td>RANGO APROXIMADO</td></tr>
<tr><td rowspan="8">TIPOS PRIMITIVOS
(sin métodos; no son objetos; no necesitan una invocación para ser creados)</td><td>byte</td><td>Entero</td><td>1 byte</td><td>-128 a 127</td></tr>
<tr><td>short</td><td>Entero</td><td>2 bytes</td><td>-32768 a 32767</td></tr>
<tr><td>int</td><td>Entero</td><td>4 bytes</td><td>$2*10^9$</td></tr>
<tr><td>long</td><td>Entero</td><td>8 bytes</td><td>Muy grande</td></tr>
<tr><td>float</td><td>Decimal simple</td><td>4 bytes</td><td>Muy grande</td></tr>
<tr><td>double</td><td>Decimal doble</td><td>8 bytes</td><td>Muy grande</td></tr>
<tr><td>char</td><td>Carácter simple</td><td>2 bytes</td><td>---</td></tr>
<tr><td>boolean</td><td>Valor true o false</td><td>1 byte</td><td>---</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="10">TIPOS OBJETO
(con métodos, necesitan una invocación para ser creados)</td><td>Tipos de la biblioteca estándar de Java</td><td colspan="3">String (cadenas de texto)
Muchos otros (p.ej. Scanner, TreeSet, ArrayList...)</td></tr>
<tr><td>Tipos definidos por el programador / usuario</td><td colspan="3">Cualquiera que se nos ocurra, por ejemplo Taxi, Autobus, Tranvia</td></tr>
<tr><td>arrays</td><td colspan="3">Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos.</td></tr>
<tr><td rowspan="7">Tipos envoltorio o wrapper (Equivalentes a los tipos primitivos pero como objetos.)</td><td colspan="3">Byte</td></tr>
<tr><td colspan="3">Short</td></tr>
<tr><td colspan="3">Integer</td></tr>
<tr><td colspan="3">Long</td></tr>
<tr><td colspan="3">Float</td></tr>
<tr><td colspan="3">Double</td></tr>
<tr><td colspan="3">Character</td></tr>
<tr><td colspan="3">Boolean</td></tr>
</table>

Los datos “booleanos” ayudan a comprobar condiciones de forma más compacta y legible:

```
import java.util.Scanner;

public class Booleano {
  public static void main(String [] args) {

    int dato;
    boolean todoCorrecto;
    Scanner teclado = new Scanner(System.in);

    do {
      System.out.print("Introduce un dato del 0 al 10: ");
      dato = teclado.nextInt();
      todoCorrecto = (dato >= 0) && (dato <= 10);
      if  (!todoCorrecto) System.out.println("No es válido!");
    } while (!todoCorrecto);

    System.out.println("Terminado!");
  }
}
```

Y los “char” se usan para manipular letras individuales:

```
public class Char1 {
  public static void main(String [] args) {

    char letra1, letra2;
    letra1 = 'a';
    letra2 = 'b';

    System.out.print("La primera letra es : ");
    System.out.println(letra1);

    System.out.print("La segunda letra es : ");
    System.out.println(letra2);

  }
}
```

Las cadenas de texto son un tipo de datos más complejo (un “objeto”, por eso su nombre se escribe en mayúsculas) y se puede realizar muchas operaciones con ellas.

Un ejemplo de algunas es:

```
import java.util.Scanner;

public class EjemploStrings {
  public static void main(String [] args) {

   // Forma "sencilla" de dar un valor
   String texto1 = "Hola";

   // Declarar y dar valor usando un "constructor"
   String texto2 = new String("Prueba");

   // Declarar sin dar valor
   String resultado;

   // Manipulaciones básicas
   System.out.print("La primera cadena de texto es: ");
   System.out.println( texto1 );

   resultado = texto1 + texto2;
   System.out.println("Si concatenamos las dos: " + resultado);

   resultado = texto1 + 5 + " " + 23.5 + '.';
   System.out.println("Podemos concatenar varios: " + resultado);
   System.out.println("La longitud de la segunda es: " + texto2.length() );
   System.out.println("La segunda letra de texto2 es: " + texto2.charAt(1) );

   // En general, las operaciones no modifican la cadena
   texto2.toUpperCase();
   System.out.println("texto2 no ha cambiado a mayúsculas: " + texto2 );
   resultado = texto2.toUpperCase();
   System.out.println("Ahora sí: " + resultado );

   // Podemos extraer fragmentos
   resultado = texto2.substring(2,5);
   System.out.println("Tres letras desde la posición 2: " + resultado );

   // Y podemos comparar cadenas
   System.out.println("Comparamos texto1 y texto2: "+texto1.compareTo(texto2));
   if (texto1.compareTo(texto2) < 0)
      System.out.println("Texto1 es menor que texto2");

   // Finalmente, pedimos su nombre completo al usuario
   System.out.print("¿Cómo te llamas? ");
   Scanner teclado = new Scanner(System.in);
   String nombre = teclado.nextLine();
   System.out.println("Hola, " + nombre);

   // O podemos bien leer sólo la primera palabra
   System.out.print("Teclea varias palabras y espacios... ");
   String primeraPalabra = teclado.next();
   System.out.println("La primera es " + primeraPalabra);

  }
}
```

8. Arrays

Para guardar una serie de objetos, todos los cuales son del mismo tipo y cuya cantidad conocemos de antemano, emplearemos un **array**, y usaremos corchetes para acceder a cada uno de los elementos:

```
public class Array1 {
  public static void main(String [] args) {

    double[] a = { 10, 23.5, 15, 7, 8.9 };
    double total = 0;
    int i;

    for (i=0; i<5; i++)
      total += a[i]; // Es lo mismo que total = total + a[i];

    System.out.println( "La media es:" );
    System.out.println( total / 5 );
  }
}
```

En general, para recorrer un array es habitual declarar la variable dentro del “for”, así como usar la propiedad “.length” para saber su tamaño, en vez de emplear “números mágicos”:

```
public class Array2 {
  public static void main(String [] args) {

    double[] datos = { 10, 23.5, 15, 7, 8.9 };

    for (int i=0; i<datos.length; i++)
      System.out.print(datos[i]+" ");
  }
}
```

Además, desde Java versión 5 existe una notación adicional para la orden “for” (similar al “foreach” de otros lenguajes), que permite obtener elementos de un array (y de algunas de las estructuras dinámicas que veremos dentro de poco) de forma más compacta:

```
public class Array3 {
  public static void main(String [] args) {

    double[] datos = { 10, 23.5, 15, 7, 8.9 };

    for (double dato: datos)
       System.out.print(dato+" ");
  }
}
```

Los arrays bidimensionales o de n dimensiones se declaran usando múltiples corchetes:

```
public class ArrayBi {
  public static void main(String [] args) {

  int[][] datos = new int[2][2];

  datos[0][0] = 5;
  datos[0][1] = 1;
  datos[1][0] = -2;
  datos[1][1] = 3;

  // +----+----+
  // |  5 |  1 | // fila 0
  // +----+----+
  // | -2 |  3 | // fila 1
  // +----+----+

  for (int i=0; i<2; i++)
    for (int j=0; j<2; j++)
      System.out.println("El dato "+i+","+j+" vale "+datos[i][j]);
  }
}
```

9. Funciones

Las funciones, al igual que en los lenguajes que derivan de C, pueden devolver un valor de un cierto tipo, o ser de tipo "**void**" si no devuelven valor. Sus parámetros se indican entre paréntesis, precedidos cada uno de ellos por su tipo de datos. Si el programa principal no va a crear objetos (que es lo habitual en los programas sencillos), las funciones deberán declararse como "**static**":

```
public class Circulo {

  public static double superfCirculo(int radio) {

    double superf = 3.1415926535 * radio * radio;
    return superf;
  }

  public static void main(String [] args) {
    System.out.println("La superficie de un círculo con radio=3 es: ");
    System.out.println( superfCirculo(3) );
  }
}
```

Como curiosidad un poco más avanzada, **Java no permite el paso de parámetros por referencia.** Las variables de tipos sencillos se pasan por valor y en el caso de los objetos, se crea una copia antes de llamar a la función.

10. Estructuras dinámicas

Existen multitud de estructuras en el lenguaje Java que pueden resultar útiles cuando se debe almacenar varios datos pero no se conoce su cantidad.

Una de las más habituales son los "**ArrayList**", que se manejan de manera parecida a un array pero son capaces de crecer de forma arbitraria.

Algunos de sus métodos son: **add, size, get, remove, contains.**

```
import java.util.ArrayList;   // Para ArrayList
import java.util.Collections; // Para ordenar ArrayList

public class ArrayList1 {
  public static void main( String[] args ) {
    ArrayList<String> datos = new ArrayList<String>();

    // Añadimos datos
    datos.add("hola");
    datos.add("adios");
    datos.add("hasta luego");

    // Ordenamos
    Collections.sort(datos);

    // Mostramos el primero... si existe
    if (datos.size() > 0)
      System.out.println("Primer dato: " + datos.get(0));

    // Borramos el primero
    datos.remove("adios");

    // Buscamos uno
    if (datos.contains("hola"))
      System.out.println("Aparece \"hola\"");

    // Y mostramos todos
    System.out.println("Contenido actual:");
    for (String unDato : datos)
      System.out.println(unDato);
  }
}
```

Si se desea directamente que se trate de una lista ordenada y sin duplicados, será mejor usar un "**TreeSet**":

```
import java.util.TreeSet;

public class TreeSet1 {
  public static void main( String[] args ) {

    TreeSet<String> datos = new TreeSet<String>();

    // Añadimos datos
    datos.add("hola");
    datos.add("adios");
    datos.add("hasta luego");
    datos.add("hola");

    // Y mostramos todos
    System.out.println("Contenido actual:");
    for (String unDato : datos)
      System.out.println(unDato);
  }
}
```

También existen colas (**Queue**), a las que se añade con ".**add**" y de las que se obtiene el primer elemento con ".**poll**()" o se consulta sin extraerlo con ".**peek**()", y pilas (**Stack**), en las que se añade con ".**push**" y se obtiene con ".**pop**()" (o se mira con .**peek**()).

Finalmente, las tablas Hash reciben el nombre de "**HashMap**". Algunos de sus métodos son: **put, containsKey, containsValue, get, size, isEmpty, keySet**.

```
import java.util.HashMap;
import java.util.Iterator;

public class HashMap1 {
  public static void main( String[] args ) {

    HashMap<Integer,String> tabla = new HashMap<Integer,String>();

    // Añadimos datos
    tabla.put(1, "uno");
    tabla.put(3, "tres");
    tabla.put(5, "cinco");

    if (tabla.containsKey(5))
      System.out.println("Aparece la clave 5");

    if (tabla.containsValue("Tres"))
      System.out.println("Aparece el valor Tres");

    int clave = 1;
    String valor = tabla.get(clave);
    System.out.println("El valor para 1 es "+valor);

    System.out.println("Tamaño de la tabla: " + tabla.size());

    if (tabla.isEmpty())
      System.out.println("Está vacía");

    Iterator<Integer> iterador = tabla.keySet().iterator();
    while (iterador.hasNext()) {
      int claveActual = iterador.next();
      System.out.println("Clave: " + claveActual +
            " valor: " + tabla.get(claveActual));
    }

  }
}
```

11. Clases

Todo programa en Java es una **clase**.

Unas clases pueden heredar de otras con la palabra "**extends**".

En principio, por legibilidad del fuente y facilidad para localizar un cierto fragmento de código, **cada clase debería estar en su propio fichero**. Si se desea tener varias clases en un mismo fichero (que no es recomendable), solo una de ellas podrá ser pública y ésta deberá tener el mismo nombre que el fichero. En caso de que exista un método "**main**", la clase que lo contiene es la que deberá ser pública.

```
class Clase {
  protected int x;

  // Constructor
  public Clase() {
    x = 5;
  }

  // Método adicional
  public void escribir() {
    System.out.println("x vale " + x);
  }
}


class ClaseDerivada extends Clase {

  public ClaseDerivada() {
    x = x + 3;
  }

}


public class PruebaDeClases {
  public static void main( String[] args ) {

    ClaseDerivada objeto = new ClaseDerivada();
    objeto.escribir();

  }
}
```

12. Interfaces

Una "**interfaz**" es básicamente un tipo un tanto especial de clase abstracta, en la que se pueden declarar métodos pero no implementarlos.

- Los atributos, en caso de tenerlos, se consideran "**static**" (iguales para todos los objetos de esa clase) y "**final**" (constantes, no modificables).
- Las clases que "implementen" esa interfaz se comprometen a especificar cómo será el comportamiento de esos métodos.
- Una clase puede implementar varias interfaces, lo que supone poder hacer algo parecido a la herencia múltiple, que permiten lenguajes como C++ pero que no permite Java.

Un ejemplo de su uso podría ser:

```
interface Saludador {
  void saludar();
}

interface Sumador {
  int datoInterno = 5;

  void sumar(int n);
}

class Clase1 implements Saludador {
  public void saludar() {
    System.out.println("Hola!");
  }
}

class Clase2 implements Saludador, Sumador {
  int dato;

  public void saludar() {
    System.out.println("Hola! Mi dato es: " + dato);
  }

  public void sumar(int n) {
    dato = datoInterno + n;
  }
}

public class Interfaces {
  public static void main(String[] args) {
    Clase1 c1 = new Clase1();
    c1.saludar();
    Clase2 c2 = new Clase2();
    c2.sumar(7);
    c2.saludar();
  }
}
```